**DAY-2 LAB PROGRAMS**

S.Kangana Sri

192210006

CSA1760-Artificial Intelligence

**25.Monkey Banana Problem**

% Define the initial state

% state(MonkeyPosition, MonkeyHasBanana, BoxPosition)

% MonkeyPosition and BoxPosition can be 'floor', 'on\_box', or specific locations like 'room\_center'.

% MonkeyHasBanana is 'yes' or 'no'.

initial\_state(state(at\_door, no, room\_center)).

% Define the goal state

goal\_state(state(\_, yes, \_)).

% Define the actions that can be performed by the monkey

% move(MonkeyPosition, TargetPosition)

action(state(MonkeyPosition, no, BoxPosition), move(TargetPosition), state(TargetPosition, no, BoxPosition)) :-

MonkeyPosition \= TargetPosition.

% push(BoxPosition, TargetPosition)

action(state(MonkeyPosition, no, BoxPosition), push(TargetPosition), state(TargetPosition, no, TargetPosition)) :-

MonkeyPosition = BoxPosition,

BoxPosition \= TargetPosition.

% climb(BoxPosition)

action(state(BoxPosition, no, BoxPosition), climb, state(on\_box, no, BoxPosition)).

% grab\_banana()

action(state(on\_box, no, BoxPosition), grab\_banana, state(on\_box, yes, BoxPosition)).

% Solve the problem by defining a path from the initial state to the goal state

solve :-

initial\_state(InitialState),

goal\_state(GoalState),

path(InitialState, GoalState, []).

% Find the path to the goal state using a recursive strategy

path(State, State, \_) :-

write('Goal reached: '), write(State), nl.

path(CurrentState, GoalState, Visited) :-

action(CurrentState, Action, NewState),

\+ member(NewState, Visited),

write('Action: '), write(Action), nl,

write('New State: '), write(NewState), nl,

path(NewState, GoalState, [NewState | Visited]).

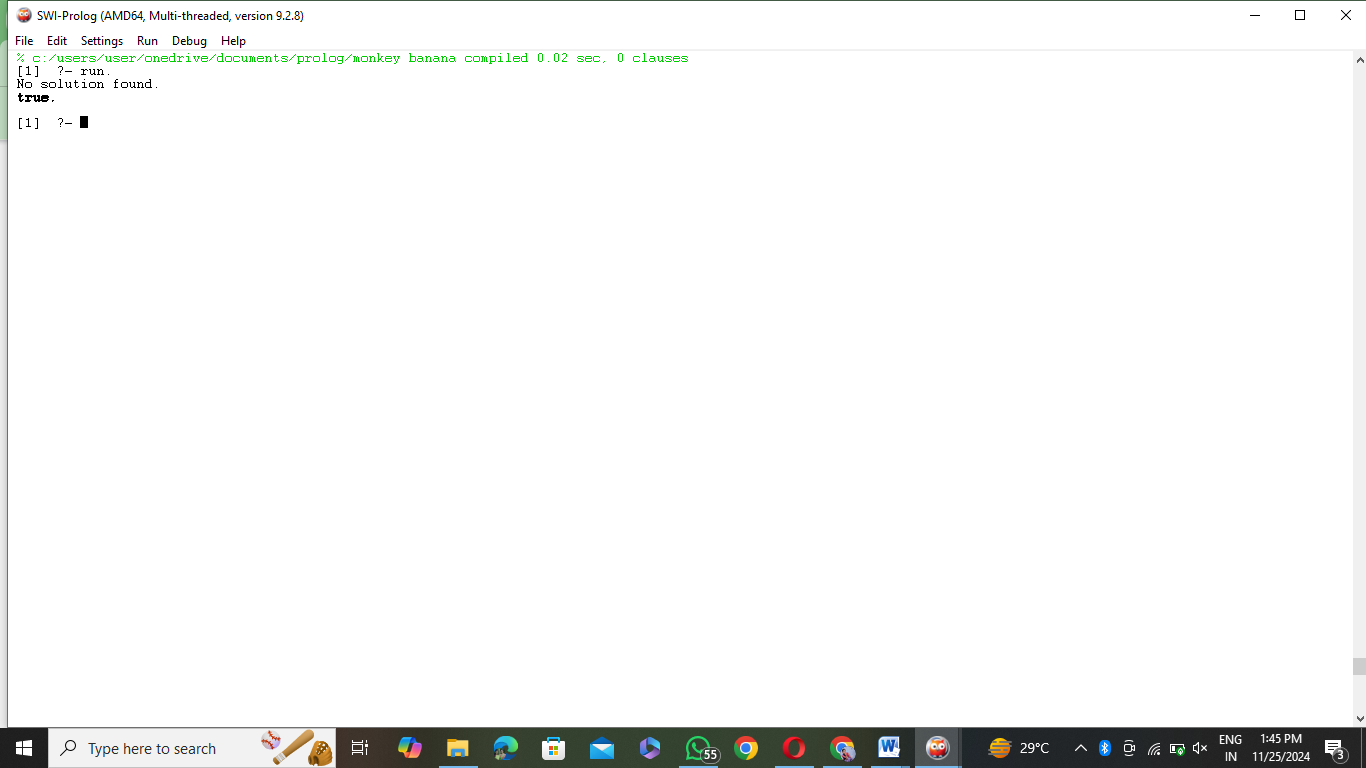
% Entry point for the program

run :-

solve;

write('No solution found.').

**Output:**

****

**26.Fruits and its colour**

% Facts: fruit(FruitName, Color)

fruit(apple, red).

fruit(banana, yellow).

fruit(grape, purple).

fruit(grape, green).

fruit(orange, orange).

fruit(kiwi, brown).

fruit(pear, green).

fruit(strawberry, red).

fruit(lemon, yellow).

% Rule: find\_fruit\_by\_color(Color, Fruit)

% This finds all fruits with a specific color.

find\_fruit\_by\_color(Color, Fruit) :-

fruit(Fruit, Color).

% Rule: find\_color\_by\_fruit(Fruit, Color)

% This finds the color of a specific fruit.

find\_color\_by\_fruit(Fruit, Color) :-

fruit(Fruit, Color).

% Queries:

% 1. Find all fruits of a specific color:

% ?- find\_fruit\_by\_color(red, Fruit).

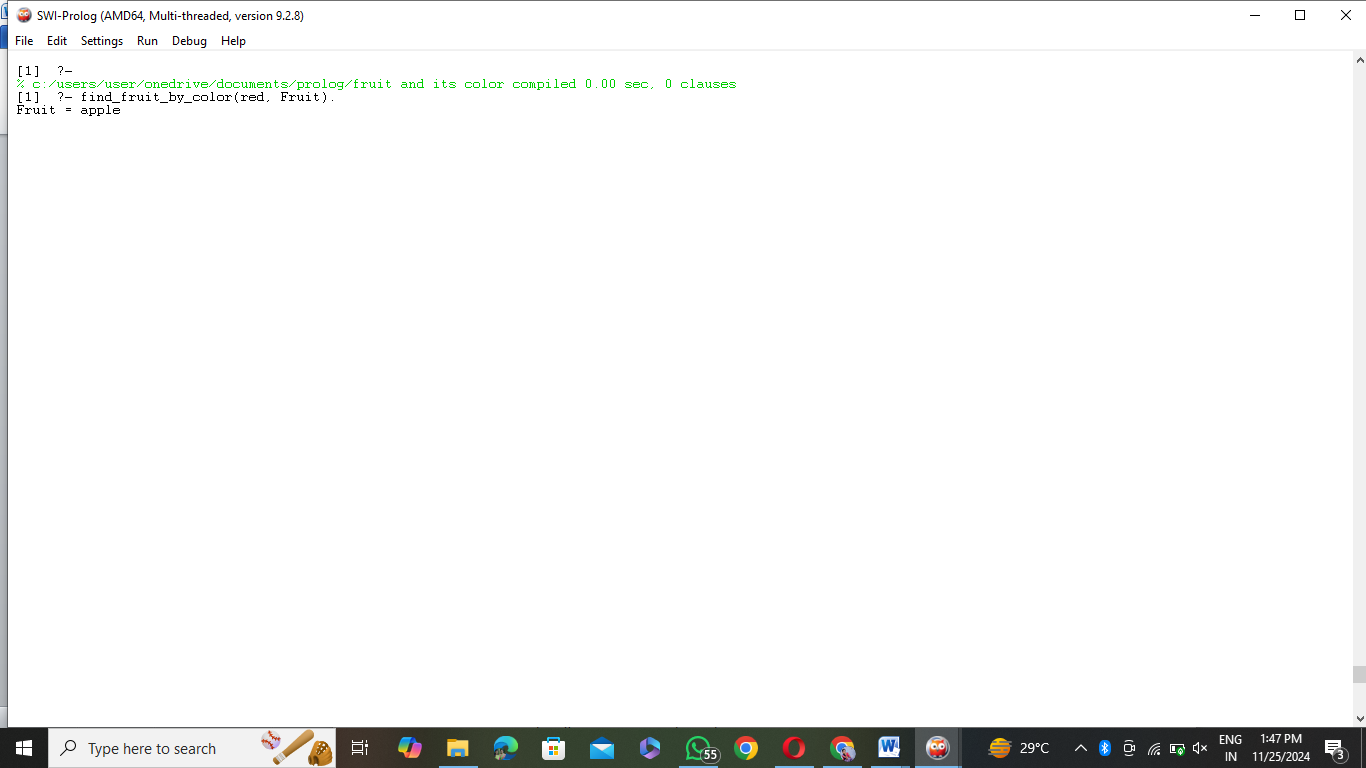
% 2. Find the color of a specific fruit:

% ?- find\_color\_by\_fruit(apple, Color).

% 3. Find all fruits and their colors:

% ?- fruit(Fruit, Color).

**Output:**

****

**27.Breadth First Search Algorithm**

% Define the graph as edges with heuristic values

% edge(StartNode, EndNode, Cost).

edge(a, b, 1).

edge(a, c, 4).

edge(b, d, 2).

edge(c, d, 3).

edge(b, e, 5).

edge(d, f, 1).

edge(e, f, 2).

% Define the heuristic values for each node

% heuristic(Node, Value).

heuristic(a, 6).

heuristic(b, 4).

heuristic(c, 5).

heuristic(d, 2).

heuristic(e, 3).

heuristic(f, 0). % Goal node has a heuristic of 0

% Best-First Search implementation

best\_first\_search(Start, Goal) :-

heuristic(Start, H),

search([[Start, H]], Goal, []).

% Base case: Reached the goal

search([[Node, \_] | \_], Node, \_) :-

write('Goal reached: '), write(Node), nl.

% Recursive case: Expand the current node

search([[Node, \_] | RestQueue], Goal, Visited) :-

findall([NextNode, H],

(edge(Node, NextNode, \_), \+ member(NextNode, Visited), heuristic(NextNode, H)),

Children),

append(RestQueue, Children, NewQueue),

sort(2, @=<, NewQueue, SortedQueue), % Sort by heuristic values

write('Visiting: '), write(Node), nl,

write('Queue: '), write(SortedQueue), nl,

search(SortedQueue, Goal, [Node | Visited]).

% Entry point for the program

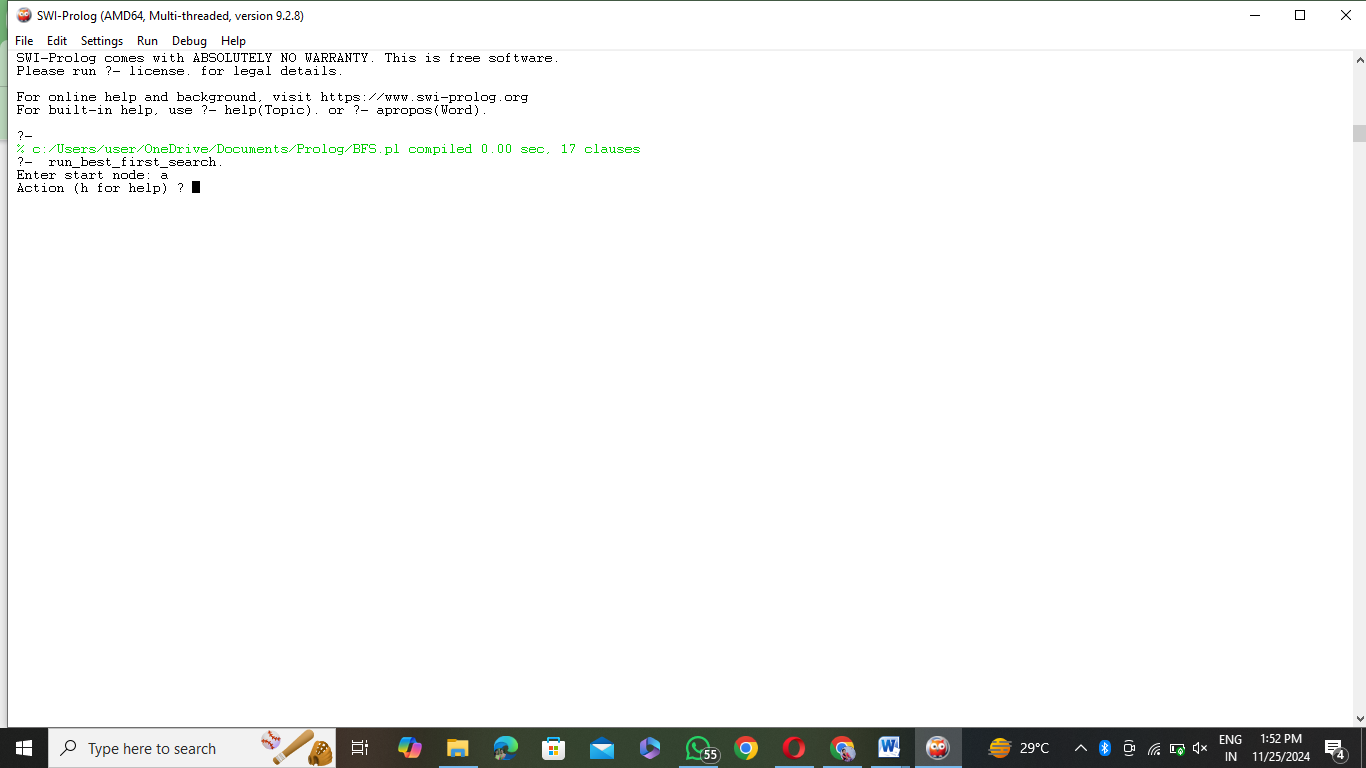
run\_best\_first\_search :-

write('Enter start node: '), read(Start),

write('Enter goal node: '), read(Goal),

best\_first\_search(Start, Goal).

**Output:**

****

**28.Medical Diagnosis**

% Facts: disease(Symptom, Disease)

disease(fever, flu).

disease(cough, flu).

disease(sore\_throat, flu).

disease(headache, flu).

disease(fever, malaria).

disease(chills, malaria).

disease(sweating, malaria).

disease(rash, chickenpox).

disease(itching, chickenpox).

disease(fever, typhoid).

disease(abdominal\_pain, typhoid).

disease(diarrhea, typhoid).

% Facts: treatment(Disease, Treatment)

treatment(flu, 'Rest, fluids, and over-the-counter medicines').

treatment(malaria, 'Antimalarial drugs and medical supervision').

treatment(chickenpox, 'Calamine lotion, rest, and antihistamines').

treatment(typhoid, 'Antibiotics and hydration therapy').

% Diagnosis Rule

diagnose(Disease) :-

write('Enter symptoms separated by commas (e.g., fever,cough): '), nl,

read(Symptoms),

findall(D, (member(Symptom, Symptoms), disease(Symptom, D)), PossibleDiseases),

list\_to\_set(PossibleDiseases, UniqueDiseases), % Eliminate duplicates

write('Possible diseases based on symptoms: '), write(UniqueDiseases), nl,

suggest\_treatment(UniqueDiseases).

% Suggest Treatment Rule

suggest\_treatment([]) :-

write('No matching disease found. Please consult a doctor.'), nl.

suggest\_treatment([Disease | Rest]) :-

treatment(Disease, Treatment),

write('For '), write(Disease), write(', suggested treatment is: '), write(Treatment), nl,

suggest\_treatment(Rest).

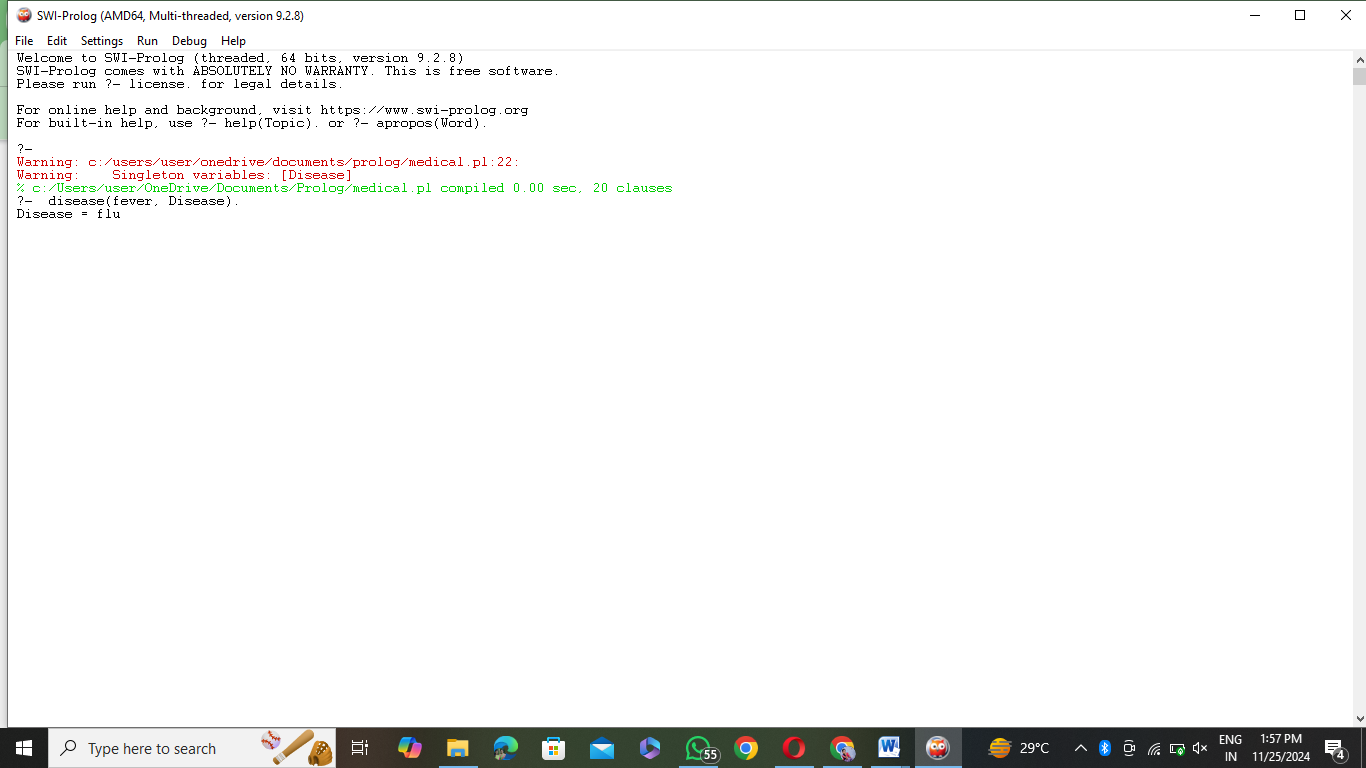
% Entry point

start :-

write('--- Welcome to the Medical Diagnosis System ---'), nl,

diagnose(\_).

**Output:**

****

**29.Forward Chaining Algorithm**

% Facts

fact(weather\_is\_rainy).

fact(ground\_is\_wet).

% Rules

rule(flash\_flood\_warning, [weather\_is\_rainy, ground\_is\_wet]).

rule(umbrella\_advice, [weather\_is\_rainy]).

% Forward Chaining Algorithm

forward\_chaining :-

findall(Conclusion, infer(Conclusion), Conclusions),

write('Inferred conclusions: '), nl,

print\_conclusions(Conclusions).

% Inference logic: Derive conclusions based on satisfied rules

infer(Conclusion) :-

rule(Conclusion, Conditions),

maplist(fact, Conditions), % Check if all conditions are facts

\+ fact(Conclusion), % Avoid duplication

assert(fact(Conclusion)). % Add the conclusion as a fact

% Helper to print conclusions

print\_conclusions([]).

print\_conclusions([H|T]) :-

write(H), nl,

print\_conclusions(T).

% Query to run forward chaining

start\_forward :-

write('Running Forward Chaining...'), nl,

forward\_chaining.

**Output:**

**30.Backward Chaining:**

% Facts

fact(weather\_is\_rainy).

fact(ground\_is\_wet).

% Rules

rule(flash\_flood\_warning, [weather\_is\_rainy, ground\_is\_wet]).

rule(umbrella\_advice, [weather\_is\_rainy]).

% Backward Chaining Logic

backward\_chaining(Goal) :-

prove(Goal, []).

% Prove the goal using recursion

prove(Goal, \_) :-

fact(Goal),

write('Proved: '), write(Goal), nl.

prove(Goal, History) :-

rule(Goal, Conditions),

\+ member(Goal, History), % Avoid circular reasoning

prove\_all(Conditions, [Goal | History]),

write('Proved: '), write(Goal), nl.

% Prove all conditions in a list

prove\_all([], \_).

prove\_all([H|T], History) :-

prove(H, History),

prove\_all(T, History).

% Query to run backward chaining

start\_backward :-

write('Enter goal: '), nl,

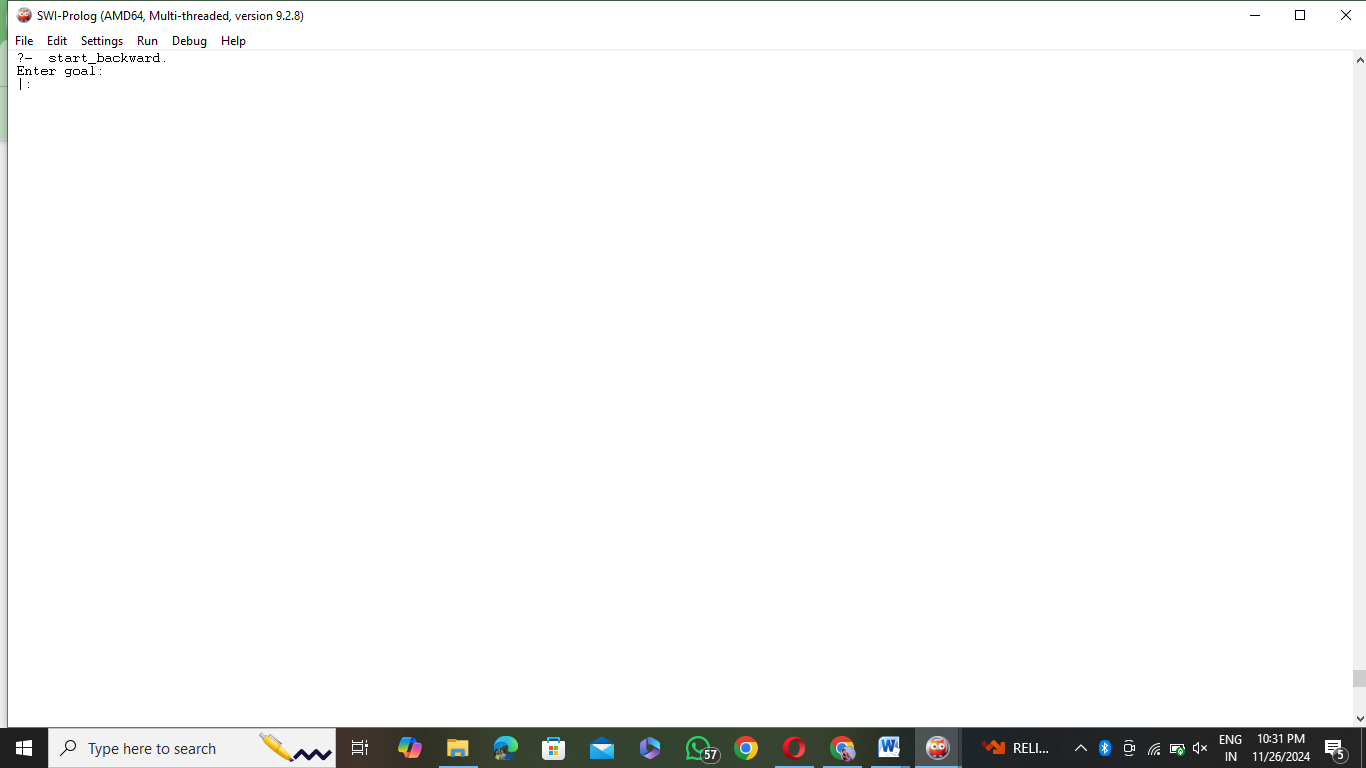
read(Goal),

( backward\_chaining(Goal) ->

write('Goal is satisfied: '), write(Goal), nl

; write('Goal cannot be satisfied.'), nl).

**Output:**

****